A. Shirt Packing

John and Alan were packing for a trip to the World Finals and discussing how many shirts they should take. John said that he was thinking about taking four shirts for a seven-day trip so that he could wear each of three shirts for two days and have a clean shirt for the last day, which sounded like a good idea. Alan said that he might wear each shirt for up to three days, so taking four shirts would allow him to wear one shirt the first three days, one for days four and five, and then have a clean shirt for each of the last two days. They then both came to the conclusion that computing how many consecutive days they could wear clean shirts would make for an interesting programming problem if practicality is ignored (typical for theoretical Computer Scientists) so that the number of days of the trip, the number of shirts packed, and the number of days they were willing to wear each shirt were only constrained to fit into an integer variable.

So that reasonable algorithms can be implemented without the need for arbitrarily long integers, you may assume that all values will fit in a 64-bit signed integer and that the product of the number of shirts packed and the number of days each shirt may be worn will fit in a 64-bit signed integer. However, do not assume that all values will fit in a 32-bit signed integer.

Input:

Each line of input will have three positive integer values: the length of the trip in number of days, the number of shirts that are packed, and the maximum number of days that each shirt can be worn. After the last input case will be a line of three zeros.

Output:

Print the case number and the maximum number of consecutive days at the end of the trip that a clean shirt can be worn. If not enough shirts are packed for the trip, print 0 days. Follow the given format exactly, "Case", a single space, the case number, a colon, a single space, the number of days, a single space, and the word 'day' if the number is one or 'days' if otherwise. (Pardon the scrolling of the fourth input case resulting from the narrow Sample Input margins.)

Sample Input	Sample Output
7 4 2 7 4 3 7 2 4 12345678901234 123456781234 105 0 0 0	Case 1: 1 day Case 2: 2 days Case 3: 0 days Case 4: 5935414695 days

B. Cycle Products

A permutation (an ordering of elements of a set without repetition) can be viewed as a (one-to-one and onto) function from a set to itself. For example, the permutation 4132 can be viewed as a function f such that $f(1) = 4$, $f(2) = 1$, $f(3) = 3$, and $f(4) = 2$. For this problem, we will assume our set is the set of natural numbers from 1 to some upper bound n ($n = 4$ in the example).

One popular notation for a permutation f involves writing f as product of cycles. A cycle is a sequence of numbers $a_1, a_2, \ldots, a_k$ such that $f(a_i) = a_{i+1}$ and $f(a_k) = a_1$. For this problem, we will assume that a_1 does not appear twice in the sequence so that the first time the cycle wraps back to a_1 would be at the end. We will write a cycle using the notation $(a_1 \ a_2 \ \ldots \ a_k)$

With this notation, the permutation 4132 can be broken into two cycles (1 4 2) and (3). The first, (1 4 2), indicates that $f(1) = 4$, $f(4) = 2$, and $f(2) = 1$. The second, (3), indicates that $f(3) = 3$.

For this problem, we will view a product of cycles as being performed left to right. [This is not universal notation, but makes sense to most beginners.] For example, (1 3 2) (2 1 3) means we follow the cycle (1 3 2) with the cycle (2 1 3) which means that the first cycle maps 1 to 3 and the second maps 3 to 2 so the product maps 1 to 2. Similarly, the first maps 3 to 2 and then the second maps 2 to 1 so the product maps 3 to 1. Finally, the first maps 2 to 1 and the second maps 1 to 3 so the product maps 2 to 3. This gives a product that can be written as (1 2 3).

Note that products do not always make for the same cycle structure as the start. The product (1 3 2) (1 2 3) maps all values to themselves and could be written as (1)(2)(3). Typically, cycles that only include one number, like (2), are omitted when writing products, but that causes us a problem with (1)(2)(3) which would then be the empty string and difficult to look at in context. Since this indicates a function that maps every value to itself, it is often called the identity function, so we will call it I. Note that the first example, the permutation 4132, would be written as simply (1 4 2).

Input/Output:

The problem here is to compute a product of up to ten cycles involving the numbers from 1 to 9. Each input line will be a valid product of from one to ten cycles in the format described above with exactly one space separating nonblank characters and no spaces at the beginning or end of the line. The output should be in a similar format (see below for details on labeling) with cycles listed in the order described by the following:

- If the result is the identity, simply print a capital I.
- Otherwise, start the first cycle with the smallest value that is in a cycle of length greater than one. The remainder of this cycle should be defined by the product computation. The next cycle should start with the smallest number not yet used that is in a cycle of length greater than one. Repeat until no more cycles of length greater than one are in the product.

Notice that, while the input is guaranteed to be valid, input lines are not required to be in the order defined for output lines. Any one cycle will not contain any number twice, but the order of values in the input cycles and the order of the input cycles is not restricted.

The last input case will be followed by a line containing only the word "End".

Output format: There is exactly one space after the word "Case", no space between the case number and the colon, one space after the colon, one space between all nonblank characters in the result, and no space at the beginning or end of any line.

Sample Input	Sample Output
(1 3 2) (2 1 3) (1 3) (1 2) (1 2) (1 3) (1 3 2) (1 2 3) End	Case 1: (1 2 3) Case 2: (1 3 2) Case 3: (1 2 3) Case 4: I

C. Necklace Checking

You have been employed by a necklace manufacturer to check designs for necklaces. Their necklaces are circular chains of circular links. [By "circular chain" we don't require that the necklace can be laid out in a perfect circle, just that the links form a cycle (closed loop) with no extra links that hang off of the cycle or connect links that are already connected.] The necklaces are depicted by a design in the form of a collection of the centers and radii of the links. The question is whether or not the given collection of links is connected in a circle. The links have some thickness and the designers use the outer radius to describe a link. The manufacturing process is such that the outer radius can be made to exact specification, but the inner radius cannot. Links are no more than 1/8 inch thick, so two links that overlap by at least ¼ inch are connected. To avoid manufacturing problems, designers will never give you a data set that includes two links that overlap by between 1/8 and 3/8 of an inch. Thus, if you check for greater or less than ¼ inch overlap, you will be fine.

There will be at most 100 links for each data set. At least 3 links are needed to lay out a circle.

Input:

Input is a collection of designs. Each design will be given as list of links. Each link will be given as three real numbers: the *x* and *y* coordinates of the center and the radius of the link. You may assume that all these values are positive. The end of a design will be indicated by 0 0 0. After the last design will be a line with three -1's.

Output:

Print the case number followed by "valid" or "invalid" indicating whether or not the given links join together to form a circular necklace. Follow the given format exactly, "`Case`", a single space, the case number, a colon, a single space, and the word "`valid`" or "`invalid`" with no trailing space.

Sample Input	Sample Output
10 20 12 20 1 12 40 30 12 50 20 12 40 1 12 20 30 12 0 0 0 25 10 5 10 10 12 10 25 5 25 25 12 25 25 12 0 0 0 -1 -1 -1	Case 1: valid Case 2: invalid

D. Bisecting Quadrilaterals

Given a convex quadrilateral and a slope, what is the *y*-intercept of the line with the given slope that cuts the quadrilateral in half (by area)?

Input:

Input will be a series of test cases. Each case will have the vertices of a convex quadrilateral listed in clockwise or counterclockwise order and be followed by the slope value. Each vertex will be defined on a separate line on the input by the *x* and *y* coordinates of the point. In addition, all *x* and *y* values in the test cases will be positive integers less than 1,000,000 and the slope will be an integer between -1,000,000 and +1,000,000. The last test case will be followed by a line with two -1's. There may be blank lines between cases.

Output:

Output will be the case number followed by the *y*-intercept for the line of given slope that divides the quadrilateral into two equal-area pieces. Print your *y*-intercept correct to five decimal places (rounded in the last spot). Follow the format below exactly with "`Case`", one space, the case number, a colon and one space, and the *y*-intercept. Your output should match the judge's output exactly except possibly in the case that the answer should be zero, in which case 0.00000 and -0.00000 are both acceptable. Test cases will be designed so that there is no danger of real number accuracy errors causing a different rounding to happen in the fifth decimal place.

Sample Input	Sample Output

`100 100`
`100 200`
`200 200`
`200 100`
`1`
`-1 -1`
	`Case 1: 0.00000`

E. Broken Clock can be Correct

"Even a broken clock is correct twice a day." That is a statement you hear sometimes to describe someone who has gotten a correct answer, or otherwise experienced success, basically by luck. However, this assumes that the clock is completely broken and is stopped or stuck at a particular time. If the clock is stopped at 2:47:00 for example, it will be correct at 2:47:00 a.m. and 2:47:00 p.m.

If we are interested only in how many times a day a clock is perfectly correct, and are not worried about how far off it is at other times, a stopped clock is actually a good option. For example, a clock that runs perfectly but is set a minute slow will never have the correct time. It will only be off by one minute, so it is more useful than a stopped clock, but the number of times it is exactly correct in one day is zero.

The problem here is to compute the number of times in the next 24 hours that a given clock is exactly correct. Do not count the start time, but do count a correct time that occurs exactly 24 hours from the start time. Note that two digital clocks could display the same time but not be at the same time if the difference is just a fraction of second. We assume we have an old fashioned 12-hour analog clock with a sweep-second hand that has to be in exactly the right position for the clock to have the correct time.

Input:

The correct time now, given as hour:minute:second (positive integers with no spaces around the colons), followed by the time on the clock, given as hour:minute:second (positive integers with no spaces around the colons), followed by an integer that gives the number of seconds that the clock runs fast in a minute. If the last integer is negative, it indicates that the clock is running slow by the absolute value of the integer. If this integer is zero, the clock is running perfectly in that it neither is fast nor slow. We may assume that no clock we are interested in is fast by more than an hour every minute, and that clocks do not run backwards, so that no clock is slow by more than 60 seconds every minute. The last input case will be followed by a line that is just the number 99.

Output:

Print the case number followed by the number of times in the next 24 hours that the clock is correct. If the clock is always correct, print "always" so that we do not have to define what a correct count would be in this case. Follow the format below exactly, "`Case`", one space, the case number, a colon and one space, and the answer for that case.

Sample Input	Sample Output

`12:15:27 12:15:20 0`
`12:15:27 12:15:27 0`
`12:15:27 4:20:30 -60`
`99`
	`Case 1: 0`
	`Case 2: always`
	`Case 3: 2`

F. Tumbling Tetrahedrons

We assume that everyone is familiar with six-sided dice, so we will view tetrahedrons as four-sided dice. Recall that the singular of "dice" is "die".

Picture a plane tiled with regular equilateral triangles (in the "normal" fashion so that 6 adjacent triangles meet at each vertex). Each triangle is marked with a random integer from 1 to 4. Now picture a modified die formed as a regular tetrahedron having sides of the same size as the triangles on the plane. Each of its faces is numbered, from 1 to 4. Specifically, by holding the die so that the three sides 1 to 3 are visible, the numbers are marked going clockwise. The hidden side is marked with the 4.

The plane defines a sort of maze on which the die can be rolled. The die is placed onto a designated starting triangle so that the number on the bottom of the die matches the number on the plane. The die can then be rolled along an edge to any of three adjacent triangles. However, we will define legal rolls as only those for which the new bottom of the die also matches the number on the new triangle in the plane. Your task is to determine the shortest path (assuming any path exists) on which the die can be rolled to a designated destination triangle. Length of the path is measured as number of rolls of the die. Note that a rotation of the die around its vertical access (turning the die but not changing what square the die is on), is not a valid move.

Input:

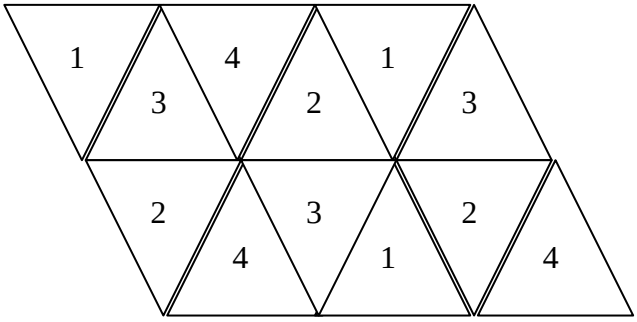
The plane of triangles is actually bounded by a parallelogram that is $r \times c$ side-lengths on each side, with the obtuse, 120 degree, corners at the lower left and upper right. The first line for each case contains the two positive integers r and c (number of rows and columns respectively). Following lines contain the $2c$ random integers for each row (noting that triangles in a row alternate having a side at the top and a side at the bottom, so it takes $2c$ of them to extend the full length of the bounding parallelogram). A row may be split among multiple lines and may be followed by blank lines.

Following the enumeration of the plane are two lines containing the row, column coordinates for the start and for the end of the desired path, respectively. We will index our plane with the lower left at (1, 1). You may assume that these are valid coordinates for a triangle in our part of the plane. There may be blank lines between cases.

End of input is indicated by both r and c being negative. Other than this line, you may assume that $1 < r, c < 50$.

To specify the starting orientation of the die, we will assume that the number at the bottom matches the number on the starting triangle and that the number on the front or the back (depending which way the starting triangle faces) matches the number on the adjacent triangle in that direction. By "front" and "back" we mean the side of the die that has a horizontal intersection with the plane so that if we are standing in the plane in the same column but with a row coordinate than the die, we see a side of the die facing us (front) or there is one that faces the opposite direction (back). (See below for examples.) We are also assuming that there is an adjacent triangle in that direction, so we will not start on the bottom facing down or on the top facing up. Thus, there will always be one valid move from the starting position.

For example, here is a simple case with $r = 2$ and $c = 3$.



As examples of front and back sides, a die starting on the lower row in the leftmost square would have a 2 on the bottom and a 3 on the back side which faces the 3 in the upper row. A die starting on the 2 in the upper row would have a 2 on the bottom and a 3 on the front side which faces the 3 in the lower row. Once we know the orientation of two sides of the die, the orientation of the other two sides is determined.

Thinking of this as on the *xy*-plane, we use the common convention of listing rows in increasing order by *x* then in increasing order by *y*. Thus, the input for this example would be

```
2 3
2 4 3 1 2 4
1 3 4 2 1 3
```

If we wanted to start at the lower left 2, location (1, 1), and move to the 2 in the upper row, location (2, 4), we could roll the die up onto the 3, then right to the 4 and right to the 2 which would give us a path of length 3. There are other ways to get there, but no shorter way, so 3 is the answer for this case.

Output:

Print the case number and the length of the shortest path, if any. Follow the format below exactly, with "`Case`", one space, the case number, a colon and one space, and the answer for that case.

Sample Input	Sample Output
2 3 2 4 3 1 2 4 1 3 4 2 1 3 1 1 2 4 2 3 2 4 3 1 2 4 1 3 1 4 1 3 1 1 2 5 2 3 2 4 3 1 2 4 1 3 2 3 1 3 1 1 2 3 -2 -2	Case 1: 3 rolls Case 2: 6 rolls Case 3: impossible

G. Fibonacci Extended

The classic definition of Fibonacci numbers is that the next number in the sequence is the sum of the previous two. For this problem, we will extend that definition to allow for adding any given number of previous values.

More precisely, we define $F(n, k, q)$ as follows:

$$F(n,k,q) = \begin{cases} 1 \quad \text{if } 0 \le n < k \\ \left(\sum_{i=1}^{k} F(n-i,k,q) \right) \bmod q \end{cases}$$

Thus, the standard Fibonacci number sequence can be viewed as $F(n, 2, q)$ if q is larger than the value we are interested in.

Input:

There will be multiple cases, each given by three integers: n, k, q (in that order, as described above). For this problem, you may assume that $0 < n < 1{,}000{,}000$, that $0 < k < 1{,}000{,}000$, and that $1 < q < 2^{30}$. The last input case will be followed by a line of three zeros.

Output:

Follow the format below exactly, "`Case`", one space, the case number, a colon and one space, and the answer for that case.

Sample Input	Sample Output

`1 2 1000`
`5 2 1000`
`5 3 10`
`0 0 0`
	`Case 1: 1`
	`Case 2: 8`
	`Case 3: 9`

H. Edit Distance in a new Dimension

The Edit Distance Problem is a classic problem in dynamic programming. In fact, it was used for this contest not that long ago. The problem this time is to extend the Edit Distance Problem to two dimensions, thus finding the minimum cost to transform one matrix into another matrix.

Recall, that the 1-D edit distance between two sequences of values is defined as the minimum number of characters that need to be replaced, added, and/or deleted from the first word to transform it into the second word.

> Consider the words "CAT" and "BAT". The two words only differ in their first character. We only need to replace the 'C' in "CAT" with a 'B' to arrive at "BAT". Therefore the 1-D edit distance is 1.
>
> The words "FLY" and "FLYING" are identical in their first three characters, but the second word has three additional characters. Adding "ING" to the first word produces the second word. The 1-D edit distance in this case is 3.
>
> Consider "GRAVE" and "GROOVY". We can perform the following substitutions in the first word: (1) 'A' → 'O', (2) 'E' → 'Y', then (3) insert the character 'O' in position 4 (after the first 'O'). Thus, the 1-D edit distance in this case is 3.

For this problem, we will use positive integers rather than characters but keep the allowed operations of change a value, insert a value, and delete a value. Each of these operations will be viewed as having a cost of 1.

Now, let us extend this to two dimensions. We want to transform one matrix of positive integers into another. We define the allowed operations and their costs as follows:

Operation	Cost
insert a row at the end	the number of numbers in the row
insert a column at the end	the number of numbers in the column
delete a row from the end	the number of numbers in the row
delete a column from the end	the number of numbers in the column
change a row in matrix 1 to match a row in matrix 2 (only allowed if same number of values)	the 1-D edit distance between the rows
change a column in matrix 1 to match a column in matrix 2 (only allowed if same number of values)	the 1-D edit distance between the columns

Note that we operate on full rows and columns. We are not allowing deleting a single value from a matrix and moving all values in the row over which would create a hole at the end of the row or moving values up in a column which would create a hole at the bottom of the column. Also, when inserting a row or column, it gives the least cost to insert a row or column that matches the other matrix, so we can implement this as deleting the corresponding row or column from the other matrix.

Input:

The input will consist of multiple cases. The first line of each case will contain four integers: the number of rows in the first matrix, the number of columns in the first matrix, the number of rows in the second matrix, and the number of columns in the second matrix. After this line will be the values in the first matrix in row-major order followed by the values in the second matrix in row-major order. End of input will be signified by having

at least one of the dimension equal to zero. Other than this end-of-input line, you may assume all dimensions are positive integers no bigger than 30.

Output:

For each input case, print the minimum cost for transforming the first matrix into the second. Follow the format below exactly, "`Case`", one space, the case number, a colon and one space, and the answer for that case.

Sample Input	Sample Output
3 3 3 3 4 3 2 3 6 2 5 1 9 3 2 7 3 3 6 5 1 9 2 2 1 4 4 3 3 6 6 7 5 3 0 0 0 0	Case 1: 4 Case 2: 6

I. What do I need to get on the Final Exam?

One typical question that students ask professors right before the final exam in a course is what score they need on the final exam in order to get a grade that they want. Your task is to write a program that automates the answer to this question.

Input:

The input will consist of multiple cases. The input for each will consist of nonnegative integers that are all at most 100 since we are assuming any test score will be at least 0 and at most 100. The first integer will be the desired average, *d*, which will be assumed to be greater than zero and less than 100. The next integer will be the number of tests (including the final exam), *k*. You may assume that *k* is at least 2 and at most 10. Then, there will be *k* integers giving the percentage weights for the tests. You may assume that the sum of the weights is 100. Finally, the scores on the $k - 1$ tests before the final are given as input. Assuming, again, that a final exam grade is at least 0 and at most 100, determine if there is a final exam grade that will achieve the desired result. If there is such a grade, the smallest such is the answer for the case. After the last valid case will be a line with a negative value.

Output:

For each input case, print the minimum final exam grade to obtain the desired average. Be sure to print a valid grade, or, if there is no valid grade that will achieve the desired average, print "impossible". Follow the format below exactly, "`Case`", one space, the case number, a colon and one space, and the answer for that case.

Sample Input	Sample Output
90 4 25 25 25 25 85 87 91 90 5 15 25 15 25 20 87 62 47 91 90 6 10 10 10 10 10 50 85 87 91 87 53 -1	Case 1: 97 Case 2: impossible Case 3: 100